Titan TTCN-3 Based Test Framework for Resource Constrained Systems

Artem Yushev [a], Manuel Schappacher and Axel Sikora

Institute of Reliable Communication and Electronics of Offenburg University of Applied Sciences, Offenburg, 77652 Germany

Abstract. Wireless communication systems more and more become part of our daily live. Especially with the Internet of Things (IoT) the overall connectivity increases rapidly since everyday objects become part of the global network. For this purpose several new wireless protocols have arisen, whereas 6LoWPAN (IPv6 over Low power Wireless Personal Area Networks) can be seen as one of the most important protocols within this sector. Originally designed on top of the IEEE802.15.4 standard it is a subject to various adaptions that will allow to use 6LoWPAN over different technologies; e.g. DECT Ultra Low Energy (ULE). Although this high connectivity offers a lot of new possibilities, there are several requirements and pitfalls coming along with such new systems. With an increasing number of connected devices the interoperability between different providers is one of the biggest challenges, which makes it necessary to verify the functionality and stability of the devices and the network. Therefore testing becomes one of the key components that decides on success or failure of such a system. Although there are several protocol implementations commonly available; e.g., for IoT based systems, there is still a lack of according tools and environments as well as for functional and conformance testing. This article describes the architecture and functioning of the proposed test framework based on Testing and Test Control Notation Version 3 (TTCN-3) for 6LoWPAN over ULE networks.

1 Introduction

With the upcoming Internet of Things (IoT), wireless communication systems and its devices more and more become part of our daily live and the overall connectivity increases rapidly since everyday objects become part of the global network. Several new wireless protocols have been developed for this purpose whereas 6LoWPAN (IPv6 over Low power Wireless Personal Area Networks) has emerged as one of the most important protocols.

The main idea of 6LoWPAN is to allow the communication using the widely spread IPV6 protocol even on resource constrained devices. In its original version it uses the IEEE802.15.4 standard as its lower transport medium with 127 bytes of payload per frame where it specifies; e.g. a compression layer [1] or fragmentation mechanism of the link-layer frames to overcome IPv6 protocol restrictions such as the Maximum Transmission Unit (MTU). However, new approaches exist to use different technologies as a basis for 6LoWPAN. One of the recent adaptions is the 6LoWPAN communication over DECT Ultra Low Energy (ULE) networks which is currently driven by the ULE Alliance.

This rapid and large development leads to the question of how such systems can be verified in general and how to guarantee conformance amongst them. Although there are lots of software implementations available (even the majority of them being available as open-source) there is still a lack of tools and environments to test such systems. This holds true for the functional testing as well as for interoperability testing. Nevertheless with the Testing and Test Control Notation version 3 (TTCN-3) a scripting language and specification exists that is commonly known and used in conformance testing of communicating systems [2]. With the help of so-called TTCN-3 compilers such as Titan TTCN-3, the target system's performance can be evaluated and, moreover, the resulting test system can verify the system under test's conformity with respective to standards or the interoperability with other implementations. Taking recent efforts of the ULE Alliance to specify and to develop 6LoDECT as a reason, the authors started to concentrate on the development of a TTCN-3 based, flexible and generic test framework to address the current lack of testing tools for wireless communication systems mainly in the field of IoT. The main goal of the testing framework is to give the possibility to create functional and compliance tests without depending on a specific communication protocol but still being flexible enough to address common challenges of different systems. The test framework was designed and specified by the authors and a reference implementation has been created with the purpose to test and verify 6LoDECT implementation recently developed by the ULE Alliance.

This paper is structured as following: Section 2 provides a brief overview of IPv6 for DECT ULE networks, as well as displays the need to perform tests. Section 3 outlines the selection of a programming

[a] Corresponding author: artem.yushev@hs-offenburg.de

language and a tool for test cases. The description of the Test System and all respective components follows in Sections 4 and 5. Finally, Section 6 gives an example of a typical test case execution routine.

2 IPv6 for DECT ULE networks

The official process of adopting the IPv6 related family of standards for IEEE 802.15.4 technologies originates in *6lowpan*, an IETF working group concluded several years ago. The results of this activity became a push for the successor's WG *6lo* to apply accumulated knowledge and define new standards for a broad range of resource constrained networks. Current 6lo's contribution extends the original scope of the *6lowpan* group to communication technologies like BACnet, Bluetooth Low Energy (BLE), ITU-T G.9959 and DECT Ultra Low Energy (ULE). The subject of the authors' work was to verify that the ULE specific *6lo* adoption fulfils the latest standard draft [3].

Usage of IPv6 functionality for any communication system obliges to comply with several requirements not directly related to the core standard, such as fragmentation/reassembly of link-layer frames and multi-hop frames delivery [4]. Due to the fact that several network technologies provide proprietary support for required features some of them can be excluded from a formal specification. This holds true for the DECT ULE standard draft which defines the usage of the underlying fragmentation/reassembly support for frames as well as a rule of thumb to restrict large Application Layer payload length. Moreover the native RPL protocol for routing in 6LoWOPAN networks is excluded from the standard proposal as not required due to the star topology of DECT ULE networks. Also a fully functional 6lo-enabled DECT ULE network should have a full spectrum of commissioning protocols like the support of multiple prefixes. In other words there is a difference in composition of generic standard 6LoWPAN implementations and the target system, thus bringing the requirement to pose a reliable and repeatable testing framework for 6lo-enabled DECT ULE systems.

To make use of existing test frameworks the Testing and Test Control Notation Version 3 (TTCN3) was selected because of its long history of usage for telecommunication protocols [2] and as a tool officially selected for 6LoWPAN plugtests driven by the European Telecommunications Standards Institute (ETSI).

3 Testing and test control notation version 3

TTCN is a scripting language with strong typing used in conformance and interoperability testing. This language—developed and maintained by ETSI—does not define specific communication interface to the System Under Test (SUT), though it provides an abstraction of the communication port towards the SUT which in turn should be implemented in another programming language. Also TTCN-3 has native support for ASN.1, IDL and XML type definitions. Figure 1 depicts a simple test case in TTCN3 notation to resolve an ETSI domain name where the Domain Name System (DNS) acts as SUT. While the current TTCN version is three (TTCN-3), the previous Tree and Tabular Combined Notation (TTCN-2) was essentially a different language sharing some common properties, nevertheless it has been a major testing language for telecommunication protocols such as UMTS, 2G, 3G, SIP, etc.

```
testcase TC resolveEtsiWww() runs on DnsClient
{
    timer t ack;
    srvPort.send(m_dnsQuestion("www.etsi.org"));
    t_ack.start(1.0);
    alt {
        []srvPort.receive(mw_dnsAnswer("172.26.1.17"))
{
            setverdict (pass);
        }
        []srvPort.receive { // any other message
            setverdict(fail);
        }
        [] t_ack.timeout {
            setverdict(inconc);
        }
    }
    t_ack.stop;
}
```

Figure 1. ETSI domain name resolver test case example in TTCN-3

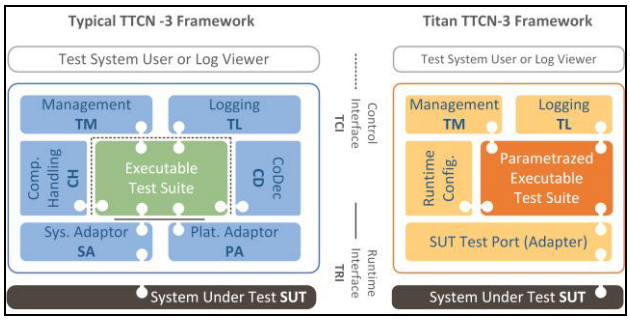

Figure 2. Typical TTCN-3 framework components and Titan TTCN-3 framework components

As with other programming languages the corresponding organization standardizes only syntax and abstract usage guide thus giving flexibility for 3rd parties to provide a compiler to generate an executable file, basically translating TTCN syntax into another programming language; e.g. C++ or Java.

Despite that ETSI formalized the TTCN-3 language syntax and made it publically available more than a decade ago there are only a few open source compilers which can offer continuous maintenance, development and supplementary tools. Probably one of the best among them is very well known within Ericsson Titan TTCN project, which a team of developers recently published on github under Eclipse Public License (EPL).

Titan TTCN uses C++ code as a supplementary language on which users should implement their communication port adapters. Moreover the Titan compiler generates C++ code from TTCN scripts and compiles afterwards the whole testing system using the preferred C++ compiler.

Figure 2 depicts the difference in structure of a typical TTCN-3 compatible compiler/framework and the Titan TTCN toolset. To be more specific, the generic TTCN approach defines the usage of two interfaces between layers. Namely, Test Control and Test Runtime interfaces,

TCI and TRI respectively, static API of which a developer can use for a test system. In contrast the Titan TTCN-3 framework specifies the usage of so-called test ports or adaptors implemented in C++. The Titan TTCN-3 code is generic: the interfaces between the test system and the system under test are specified at the level of the exchanged abstract data messages and signals. Setting up and maintaining the transport connections as well as sending/receiving "real" messages and signals are the tasks of interface adaptors. Titan's main purpose is functional testing (conformance, fuzzing, function, integration verification, end-to-end and network integration testing), and can also be used for performance testing. Titan proposes to use a C++ API for adaptors that would complete the test with the connectivity layer(s) between the test system and the system under test. However, in the authors' opinion this proven and elegant approach requires several modifications and customization for distributed test systems.

Based on the Titan TTCN toolset together with some improvements the authors have developed a test system for one specific 6lo Implementation Under Test, although the overall facility is not designed specifically for one System Under Test, but follows a more generic and abstracted approach.

4 Test system overview

4.1 System requirements

The test framework should comply with the following requirements: 1) It should be flexible enough to operate with virtual devices and with real devices without changes. Depending on the source code of a SUT, sometimes final images for different systems don't behave identically thus bringing the requirement to perform test cases on a virtual setup during the development phase and on a real setup with real devices; 2) The test system should have an option to control SUTs remotely. Sometimes it runs test cases without being physically on the same machine with the SUTs. In this case the authors consider the test framework a distributed system; 3) The test system should have an option to deliver control information and messages to the SUT without a direct usage of a specific communication interface. Due to the distributed character of the testing framework it might not be a rare case when SUT being a device have only a low data rate, serial connection as the sole communication interface. In this case, if a fully automated test system launches test cases from a remote server, it shouldn't have a direct connection with each SUT instead utilize one channel with high data rate to some multiplexing application which manages all the connections with the SUTs [5].

An inspiration for the overall design is [6]. Despite an extensive description of a Generic SUT adaptor it cannot be directly applied to the Titan based test system as the original proposal deploys TRI and TCI API written in Java.

Therefore, the authors propose a modular Communication Interface for Titan TTCN-3 [7]. Figure 4 demonstrates the generic view on the Test Framework with all possible scenarios, whereas Figure 3 depicts a more detailed comparison between different communication setups, namely LCM inter-process communication (IPC) for virtual setup and serial interface for setup with real devices. The test system architecture shares some features with the GNU Debugger (GDB) toolset architecture. GDB uses a client-server model of communication where a GDB-server connects directly to a target via the hardware debugger/programmer and the GDB-client can be located anywhere and send instructions to the IP address of the machine on which the GDB server runs.

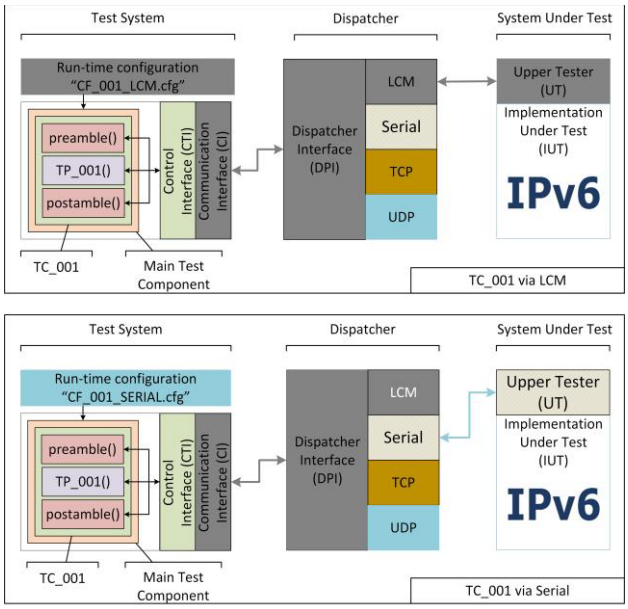

Figure 3. Test system architecture overview for 6lo over ULE networks implementation. Test System on the left side, Dispatcher in the middle and System Under Test on the right side

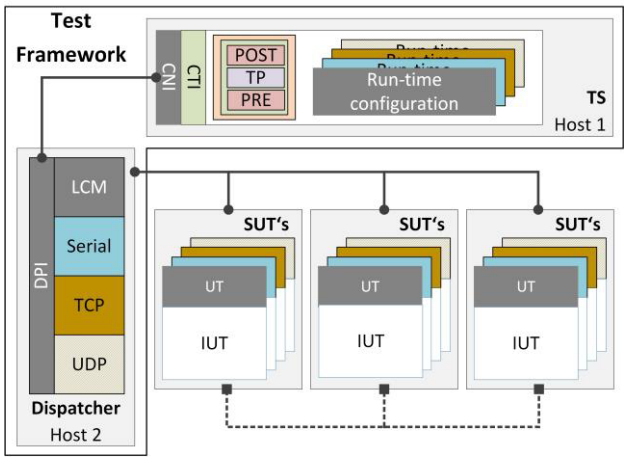

Figure 4. Test system architecture overview. Interconnection overview

In the current implementation the Test System acts in the same role as a GDB client located on the left in Figure 3, in the middle, the Dispatcher (DPI) acts as a GDB server which is located nearby the test field and finally a System Under Test is on the right side, where this SUT is a target of debugging.

4.2 General system overview

Generally *Host 1* (cf. Figure 4) controls the test cases' execution and manages the verdicts, as well as provides logging. A TCP/IP based communication link interconnects *Host 1* and *Host 2*. *Host 2* in its turn on another end—they can be separate devices or the same with different opened sockets—waiting for incoming messages. Moreover *Host 2* possesses all possible communication interfaces to the SUTs whether it be a company's proprietary or generic protocol. More precisely there are three entities in the testing framework:

1. The Test System (TS) which controls the execution of test cases.
2. The Dispatcher (DPI) which routes requests and responses from TS to SUT and backward.
3. The System Under Test combines the Upper Tester (UT), which maps commands from TS to direct API calls, and the Implementation Under Test (IUT).

4.3 Test system

The Titan compiler builds an executable TS (ETS) from the TTCN-3 test suite, called abstract test suite (ATS), adapter code and the Titan runtime library. The ETS can't be launched without a configuration file as Titan provides a very flexible runtime parametrization of the test cases; e.g. it is possible to select the exact type and properties of the connection between DPI and SUT, settings for the test cases etc.; the values of runtime parameters need not be defined in development time—though default values can be specified—but they can be provided just before the test execution session Therefore flexible test scenarios can be launched with different configurations for different setups (virtual, serial, TCP/IP or proprietary) without rebuilding the ETS.

The Titan TTCN-3 tool set has a runtime library which makes available the runtime control with the Main Controller (MC), where the latter has several tasks:

- It distributes a runtime configuration to all test components.
- It controls the execution of all test cases and generates verdicts.
- It has a failsafe engine which allows to catch runtime "errors" of a specific test case, to clean allocated resources and to proceed to the next test case.
- It logs all performed action on the system using the logging level predefined in the runtime configuration.

Given that the MC launches the Main Test Component (MTC) which controls other components including the test steps procedure, the standard routine in each test case can be divided on three phases:

1. The pre amble phase sets a remote IUT in initial and defined state, ready for a test purpose (subject of the test case) execution, also referred to as Stimulus;
2. The test purpose phase checks the behavior and message flow between nodes as well as sends extra requests if required;
3. The post amble phase brings the system in the final state possibly cleaning tracks of activity on SUT.

4.4 Dispatcher

When a test case transmits control data or a message via the CTI interface to the Dispatcher entity, the latter parses received packet, executes commands if required and dispatches data to the requested SUT. In the current version of the testing framework the Dispatcher entity is written in C++ language, as on the one hand this language follows the OOP paradigm which offers code modularity with flexibility and on the other hand, because of its high performance it minimizes a possible processing delay of data between TS and SUT.

The Dispatcher accepts the following requests from the TS:

- *Open*, to open a connection with a SUT, assign a SUT Interface Identifier (ID) and return it to the TS;
- *Close*, to close a specific connection with a SUT Interface using the SUT Interface ID received as a parameter of the command;
- *Control*, to perform managing actions with a SUT Interface as well as to forward control payloads towards a SUT using SUT Interface ID received as a parameter of the command;
- *DataOut*, to forward directly the payload to the selected SUT Interface.

Also, the Dispatcher generates responses about SUT interfaces as well as forwards data to the CTI of the TS. Possible responses from the Dispatcher are the following:

- *Opened*, to notify the TS about the open status of the SUT Interface and also to transmit a SUT Interface Identifier, which can be used for all further requests;
- *Closed*, to notify the TS about the closed connection status with the SUT Interface;
- *Status*, to transmit the response to a control request from a SUT;
- *DataIn*, to transmit a payload from a SUT to the TS.

4.5 System under test

As was indicated previously the SUT is composed of the Implementation Under Test and the Upper Tester entities. The SUT requires the UT in order to translate specific configuration commands sent from the TS into direct API calls. Here are some examples of the usage of UT:

- To reset, initialize or de-initialize the IUT.
- To read temporary characteristics of the IUT.
- To process data from the inner side to the outer or vice versa.
- To emulate user actions.

However, it is also possible to have a SUT without an Upper Tester component, although in this case a distinct component should persist nearby which allows to inject data into a common network.

5 Communication interfaces

There are two types of communication interfaces on the TS (cf. Figure 3):

- The Connection interface (CNI) links together Test System and Dispatcher interface. It establishes a stable bidirectional LCM-based connection [8].

- The Control interface (CTI) manages the SUT Interface and provides and API to the TS. Also the CTI hides the exact SUT Interface selection from the user using a simple communication protocol with fixed format and 8 types of messages: OPEN; OPENED; CLOSE; CLOSED; CONTROL; STATUS; DATAOUT; DATAIN, described previously in Sections 4.3 and 4.4.

These two pairs of interfaces play two distinct roles in the test scenario. When the control interface has the abstract static API available for the Test System by which the latter can control the SUT Interface driver running on the Dispatcher unit, the connection interface provides a possibility to deliver all the commands and data to a remote system and backward via a stable and reliable connection.

6 Test execution

Figure 6 shows a specific test setup configuration for 6lo over ULE networks. For the purpose of only 6lo implementation conformance testing, the provided library has a full network stack excluding DLL and PHY layers, instead the software establishes a UDP connection, thereby emulating the properties of a connection with other nodes of the network.

The test setup consists of the Test System with the Main Test Component (MTC), the Dispatcher and two Systems Under Test which represent a router entity and a local node. Two SUTs have an Upper Tester component to link the Test system with the Implementation Under Test, each of the UTs have a library to parse commands and translate them into direct API calls. The authors propose to have a modular approach for the UT's software architecture in order to provide developers with a library with a flexible substitution of Command or Transport engines.

Each performed test case for this setup requires at least several components: 1) The test case code in TTCN-3 language; 2) The CI (CTI and CNI) adapter implementation in C++ which will deliver the test case commands' responses as well as raw binary frames from the SUTs; 3) The Codec/Decodec (CoDec) entity which is generated automatically by the Titan TTCN-3 tool set from TTCN-3 code (cf. Figure 5).

```
/* @desc    6lowpan Frame (RFC 4944/6282) */
type union SixLo_frame
{
    SixLoUCIpv6    ucipv6,    // Uncompressed IPv6
    SixLoHC6282    hc,        // Compressed IPv6
    SixLoMesh      mesh,      // Mesh Header
    SixLoBroadcast broadcast,// Broadcast Header
    SixLoFrag1     frag1,     // Fragment Header 1
    SixLoFragN     fragN      // Fragment Header N
} with { variant "TAG(
        ucipv6,    dispatch = '01000001'B;
        hc,        dispatch = '011'B;
        mesh,      dispatch = '10'B;
        broadcast, dispatch = '01010000'B;
        frag1,     dispatch = '11000'B;
        fragN,     dispatch = '11100'B )"
}
```

Figure 5. An example of 6lo frame definition in TTCN-3 with Titan automatic coder/decoder generation

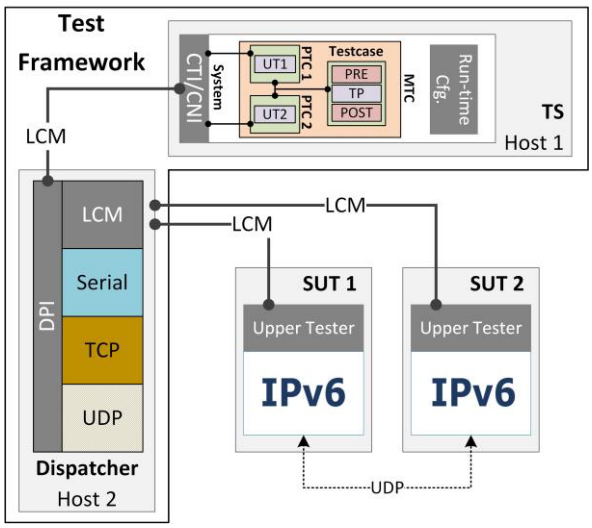

Figure 6. The overview of typical setup for one of 6lo over ULE networks test cases

A Titan test case works with abstract data; e.g, like a 6LoWPAN link-layer frame definition listed in Figure 5. That is why the CoDec entity deserialize binary data into a TTCN-3 data type; e.g. if an incoming frame starts with a *011* binary sequence the CoDec creates the *SixLo_frame* object with only one internal field, *hc*, inside which the *dispatch* field comes first, followed by the remaining payload.

References

1. Z. Shelby, and C. Bormann, "6LoWPAN: The wireless embedded Internet", Vol. **43**, John Wiley & Sons, (2011)
2. J. Grabowski, et al. "An introduction to the testing and test control notation (TTCN-3)." Comp. Net. **42**.3 (2003)
3. Transmission of IPv6 Packets over DECT Ultra Low Energy, draft-ietf-6lo-dect-ule-03, (2015)
4. M. Schappacher, et al. "A flexible, modular, open-source implementation of 6LoWPAN." IDAACS-2015, 2015 IEEE 8th Int. Conf. on, Vol. **2**, (2015)
5. A. Yushev, et al. "Extended performance measurements of scalable 6LoWPAN networks in an Automated Physical Testbed." I2MTC-215, 2015 IEEE Inter. Conf., IEEE, (2015)
6. A. Hyrkkänen, "General purpose SUT adapter for TTCN-3." MS thesis, Tampere Univ. of Tech., (2005)
7. J. Z. Szabó, and T. Csöndes. "TITAN, TTCN-3 test execution environment." Infocomm. J. **62**.1 (2007): 27-31
8. A. S. Huang, et al. "LCM: Lightweight communications and marshalling." Intelligent robots and systems (IROS), IEEE/RSJ Intl. Conf., (2010)